Noise-contrastive estimation of normalising constants and GANs

Contents

1 Fonctions génériques

1.1 Algorithme d'Hasting

Utilité : simuler selon $p_m(.,\psi)$ pour un paramètre ψ choisi.

Argument	**Type**	**Exemple**	**Indication**
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité inconnue
n	entier	100	taille de la simulation
psi	vecteur	c(0,1)	paramètres de la fonction h
h	fonction		fonction qui retourne $\overline{p_m}(.,\psi)$
Sortie	**Type**	**Exemple**	**Indication**
y	vecteur		notre échantillon simulé

```
hasting = function(x, n, psi, h){
  y = c()
  y = append(y, sample(x, 1))
  for (i in 2:n){
    y_ = rnorm(1, y[i-1], 1)
    u = runif(1)
    if ( u <=
         (h(y_,psi) * dnorm(y_, y[i-1], 1))
         /(h(y[i-1],psi) * dnorm(y[i-1], y_, 1))
    ){
      y = append(y, y_)}
    else {
      y = append(y, y[i-1])
    }
  }
  return (y)
}
```

Ci-dessous une autre version qui génère un échantillon iid.

Argument	**Type**	**Exemple**	**Indication**
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité inconnue
n	entier	100	taille de la simulation
psi	vecteur	c(0,1)	paramètres de la fonction h
h	fonction		fonction qui retourne $\overline{p_m}(.,\psi)$
ϵ	Entier	2	pas de décorrélation $\overline{p_m}(.,\psi)$

```
hasting_iid = function(x, n, psi, h, eps){
  y = c()
  y = append(y, sample(x, 1))
  for (i in 2:(n*eps)){
    y_ = rnorm(1, y[i-1], 1)
    u = runif(1)
    if ( u <=
         (h(y_,psi) * dnorm(y_, y[i-1], 1))
         /(h(y[i-1],psi) * dnorm(y[i-1], y_, 1))
    ){
      y = append(y, y_)}
    else {
      y = append(y, y[i-1])
    }
```

```
  }
  filter = y * rep(c(1,rep(0, eps-1)), n)
  return (filter[filter != 0])
}
```

1.2 MC MLE (Geyer)

Utilité : retourne une estimation des paramètres selon la méthode décrite dans le papier de Geyer.

```
mc_mle = function(x, n, psi, h){

  m = length(x)

  y = hasting(x, n, psi, h)

  L = function(theta){
    return(sum(log(h(x,theta)/h(x,psi))) - m*log(mean(h(y,theta)/h(y,psi))))
  }

  theta = optim(
    par = rep(1,length(psi)),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  return(theta)
}
```

1.3 NCE (Gutmann)

Utilité : Retourne l'estimation de la constante et des paramètres.

Argument	Type	Exemple	Indication
x	vecteur	rcauchy(100, 0, 1)	notre échantillon de densité inconnue
law_y	fonction	rnorm	fonction qui retourne un échantillon suivant la loi p_n
n	entier	100	taille de l'échantillon de bruit suivant la loi p_n
params_y	vecteur	c(0,1)	arguments de la fonction law_y
log_pm	fonction		fonction qui retourne le logarithme de la densité p_m
log_pn	fonction		fonction qui retourne le logarithme de la densité p_n
size_theta	entier	3	taille de θ, vaut habituellement 2 ou 3

```
nce = function(x, law_y, params_y, log_pm, log_pn, size_theta, n){

  y = do.call(law_y, c(list(n),params_y))

  m = length(x)

  h = function(u, theta){
    return( 1 / (1 + n/m * exp(log_pn(u) - log_pm(u, theta))))
  }

  J = function(theta){
    return( sum(log(h(x, theta))) + sum(log(1 - h(y, theta))) )
```

```
  }

  theta = optim(
    par = rep(1, size_theta),
    gr = "CG",
    control = list(fnscale=-1),
    fn = J
  )$par

  return(c(theta[-size_theta], exp(-theta[size_theta])))
}
```

1.4 Graphiques

Utilité : afficher l'histogramme pour un échantillon de données x.

```
print_hist = function(x) {
  df = data.frame(x = x)
  hist_x = ggplot(df, aes(x=x)) +
           geom_histogram(aes(y = ..density..), bins = 20, color="white", fill = "grey") +
           labs(y = "Fréquence") +
           stat_function(fun = dcauchy, args = list(location = mean(df$x), scale = sd(df$x)), size = 1,
           scale_colour_manual("Densité du bruit :", values=c(pal2[1]), labels = expression(Cauchy(m[em
           theme(legend.position="bottom", legend.box="vertical", legend.margin=margin())
  print(hist_x)
}
```

Utilité : pour NCE, afficher l'évolution des paramètres au fur et à mesure de l'augmentation de n (la dimension de l'échantillon de bruit)

```
NCE_evol_params = function(x, law_y, params_y, log_pm, log_pn, size_theta, ratio, steps, labels) {

  # Creation de l'abscisse
  m = length(x)
  N = seq(0, m*ratio, length.out = steps + 1)

  # Creation de l'ordonnée
  theta = c()
  for (n in N) {
    theta = append(theta, nce(x, law_y, params_y, log_pm, log_pn, size_theta, n))
  }

  # Formatage des données
  theta = t(rbind(matrix(theta, nrow = size_theta),N))
  df = as.data.frame(theta)
  df_melted = melt(df, id.vars = "N")

  # Plot
  plot_df = ggplot(df_melted, aes(x = N, y = value)) +
  geom_line(aes(color = variable, group = variable)) +
  geom_point(aes(color = variable, group = variable)) +
  labs(title = "Evolution des paramètres par rapport au bruit",
       x = "n (taille du bruit)",
       y = "Paramètres",
       color = "Légende") +
```

```
  scale_color_manual(labels = labels, values = c("blue", "red", "orange"))

  print(plot_df)

  #return(theta)
}
```

Note : il faudrait optimiser le temps de calcul de ces fonctions, peut-être en matriciel au lieu des boucles ou bien avec du calcul en parralèle sur CPU/GPU

2 Illustration avec la loi normale

Soit x l'échantillon de taille m obtenu selon la loi de densité inconnue p_d.

On considère ici que p_d appartient à la famille de fonctions paramétrées par $\theta = (c, \mu, \sigma)$ suivante :

$$p_m(u;\theta) = \frac{1}{Z(\mu,\sigma)} \times exp\big[-\frac{1}{2}\big(\frac{u-\mu}{\sigma}\big)^2\big] \quad \text{d'où} \quad ln(p_m(u;\theta)) = c - \frac{1}{2}\big(\frac{u}{\sigma} - \frac{\mu}{\sigma}\big)^2$$

```
pm_barre = function(u, theta){
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

log_pm = function(u,theta){
  return(theta[3] - 1/2 * (u/theta[2] - theta[1]/theta[2]) ** 2)
  # theta[1] = mu / theta[2] = sigma / theta[3] = c
}

log_pn_cauchy = function(u){
  return(log(dcauchy(u, mean(x), sd(x))))
}

log_pn_unif = function(u){
  return(log(dcauchy(u, min(x), max(x))))
}

m = 10000
n = 10000
x = rnorm(m, 2, 4)
size_theta = 3
```

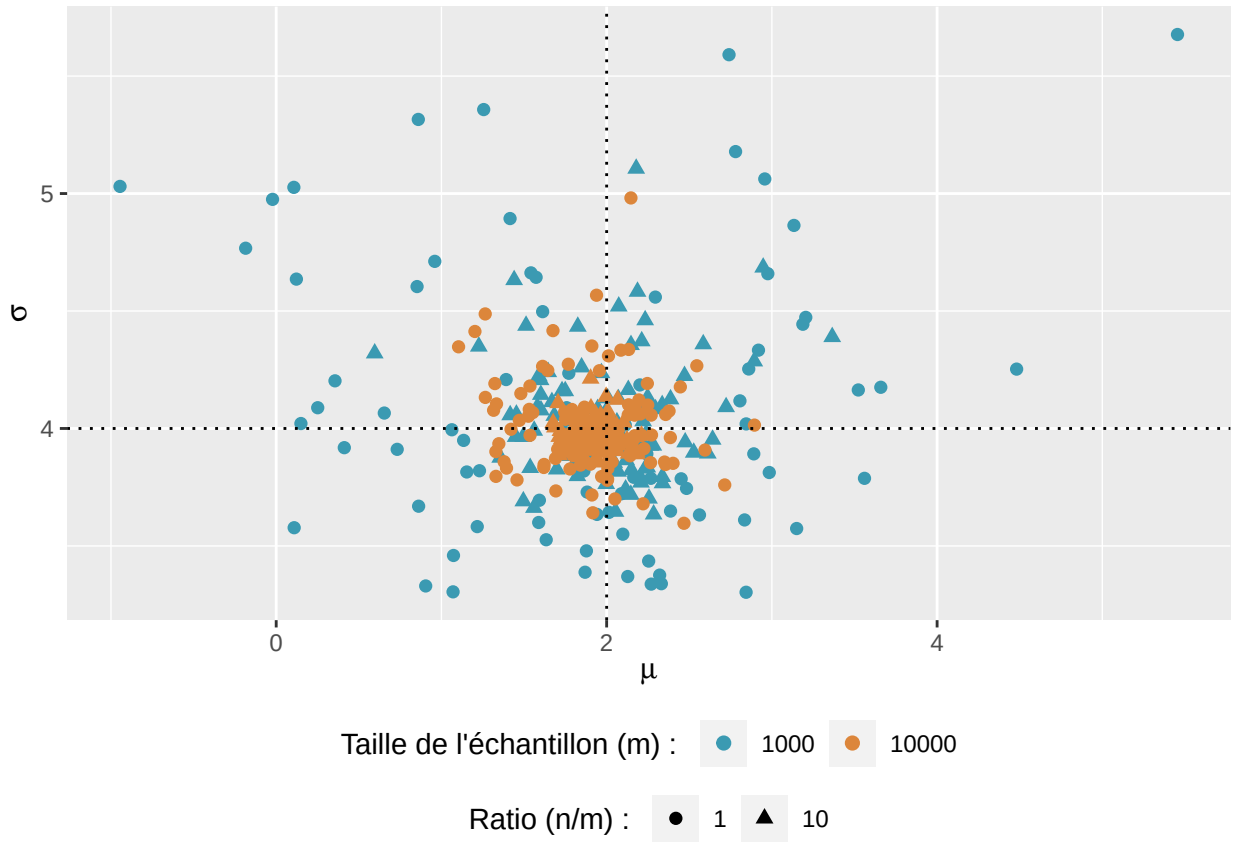
2.1 Méthode MC MLE

```
# METHODE MC MLE
mc_mle(x, n, c(10,5), pm_barre)
```

```
## [1] 2.256410 3.896519
```

Etudions l'impact de la dimension des échantillons sur la convergence des estimateurs.

```
#png('df_mcmle.png', units="cm", width=15, height=11, res=300)

ggplot(df_mcmle_filt, aes(x = param_1, y = param_2, color = size_data, shape = ratio_noise_data)) + geo
```




```
#dev.off()
```

Etudions l'impact du choix de ψ sur la convergence des estimateurs.

```
df_mcmle_psi <- read_csv("dataframes/df_mcmle_psi.csv")[,-1]
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
df_mcmle_psi = df_mcmle_psi[order(-df_mcmle_psi$ratio_alpha_psi),]

lab = list(bquote(psi == "(2,4)"), bquote(psi == "(6,12)"), bquote(psi == "(10,20)"), bquote(psi == "(1

df_mcmle_psi$ratio_alpha_psi = as.factor(df_mcmle_psi$ratio_alpha_psi)

#png('df_mcmle_psi.png', units="cm", width=15, height=11, res=300)

ggplot(df_mcmle_psi, aes(x = param_1, y = param_2, color = ratio_alpha_psi)) + geom_point(size = 2) + s
```


```
#dev.off()
```

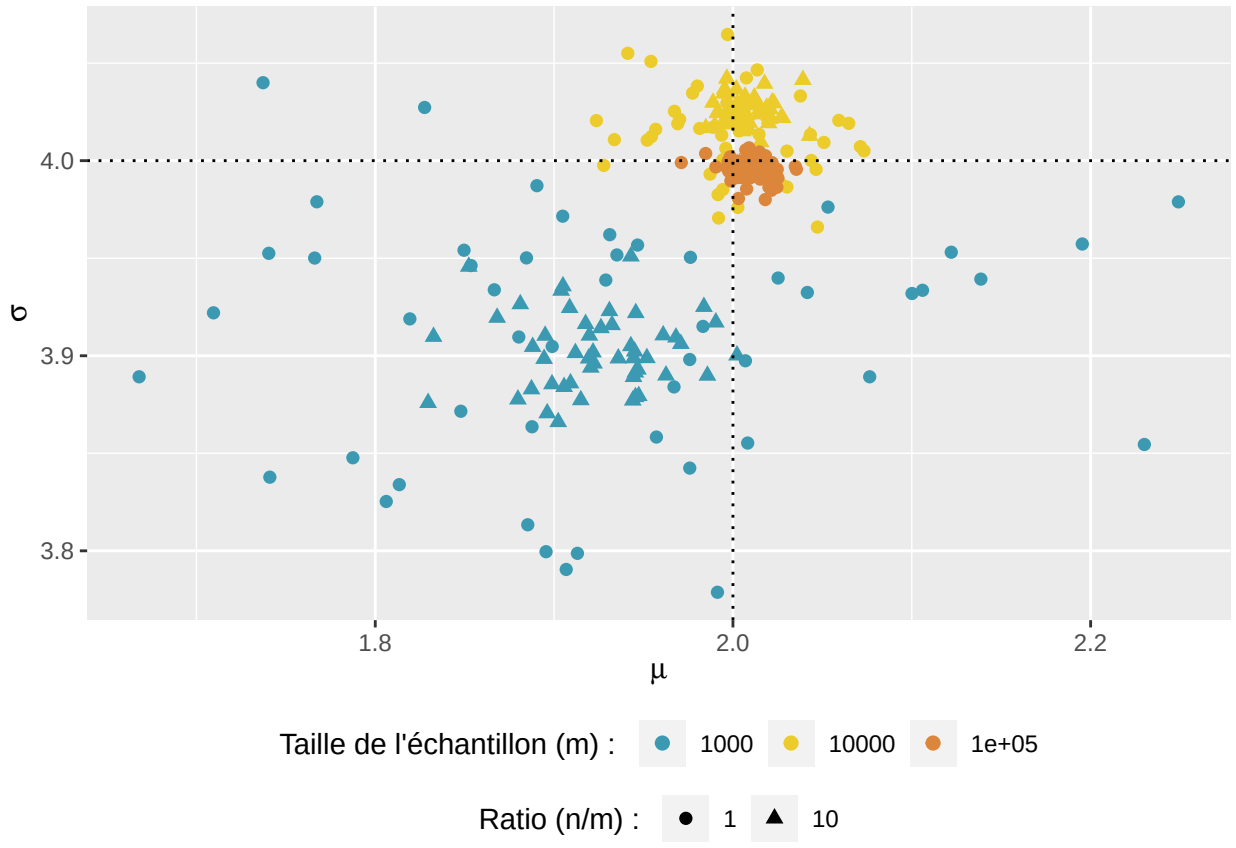
2.2 Méthode NCE

```
# METHODE NCE
nce(x, rcauchy, c(mean(x),sd(x)), log_pm, log_pn_cauchy, size_theta, n)
```

```
## [1]  2.014825  4.037277 10.144032
```

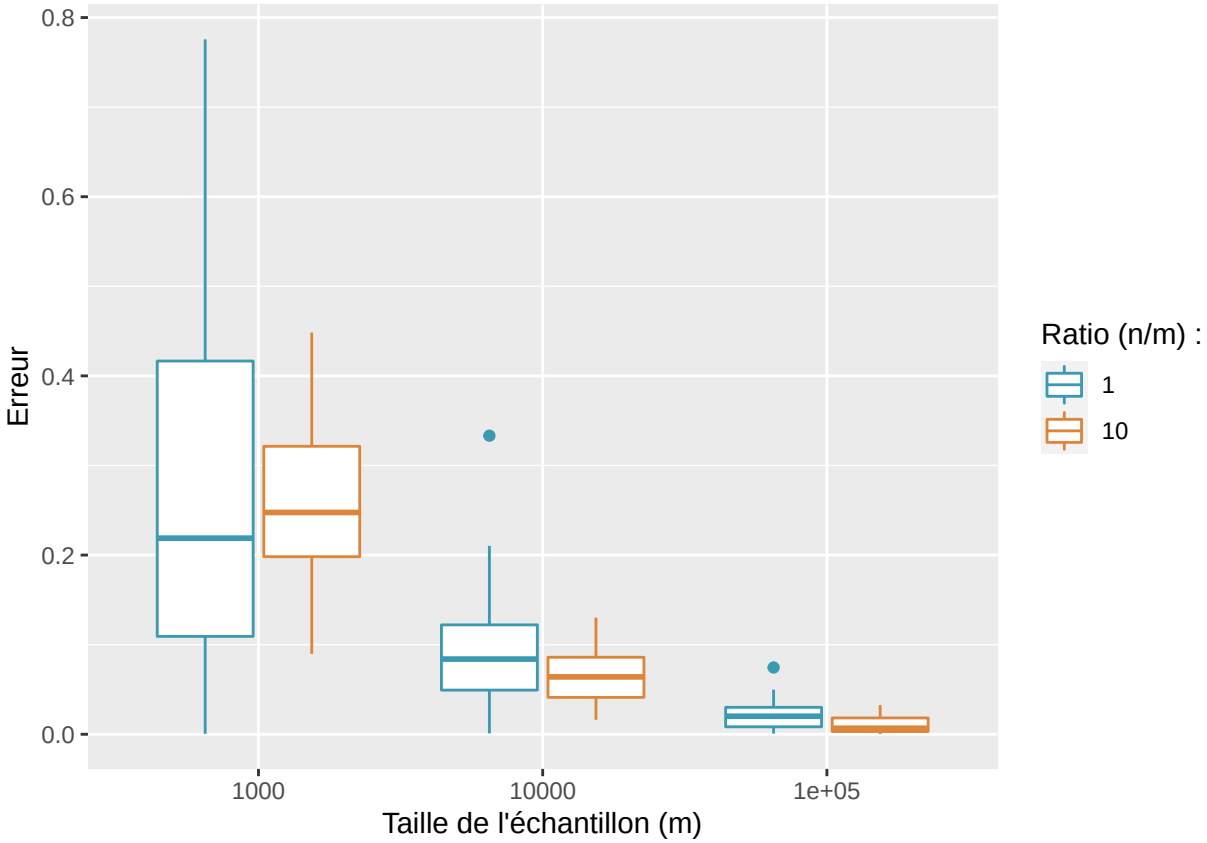
On étudie l'impact de n et m pour une loi de Cauchy.

```
#png('df_nce.png', units="cm", width=15, height=11, res=300)

ggplot(df_nce, aes(x = param_1, y = param_2, color = size_data, shape = ratio_noise_data)) + geom_point
```




```
#dev.off()
```

```
#png('df_nce_const.png', units="cm", width=15, height=11, res=300)

ggplot(df_nce, aes(x = size_data, y = const_error, color = ratio_noise_data)) + geom_boxplot() + scale_
```




```
#dev.off()
```

Compairason entre un bruit issu d'une loi de Cauchy et un bruit issu d'une loi uniforme.

3 Nouvelles approches

3.1 Bootstrap

```
#kable(bootstrap(df_nce_bootstrap, 0.05))
```

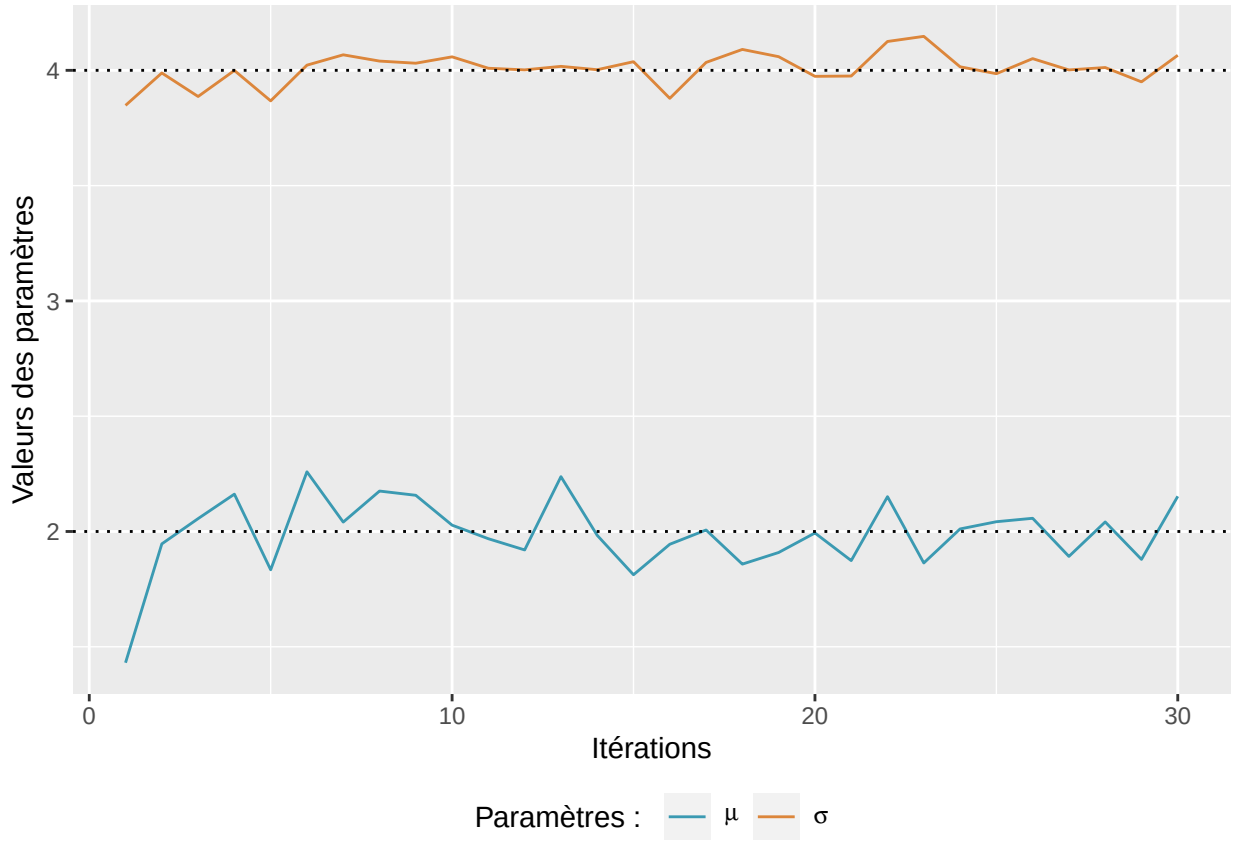
3.2 Récursivité

Utilité : améliorer récursivement la précision de l'estimation via les estimations précédentes

```
df_recurs_naif <- read_csv("dataframes/df_recurs_naif.csv")[,-1]
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
colnames(df_recurs_naif) = c("iteration", "param_1", "param_2")

lab = list(bquote(mu), bquote(sigma))

#png('df_recurs_naif.png', units="cm", width=15, height=11, res=300)

df_recurs_naif_melted = melt(df_recurs_naif, id.vars = "iteration")
ggplot(df_recurs_naif_melted, aes(x = iteration, y = value)) + geom_line(aes(color = variable, group = v
```




```
#dev.off()
```

Cette approche naïve n'améliore pas la précision de notre estimation.

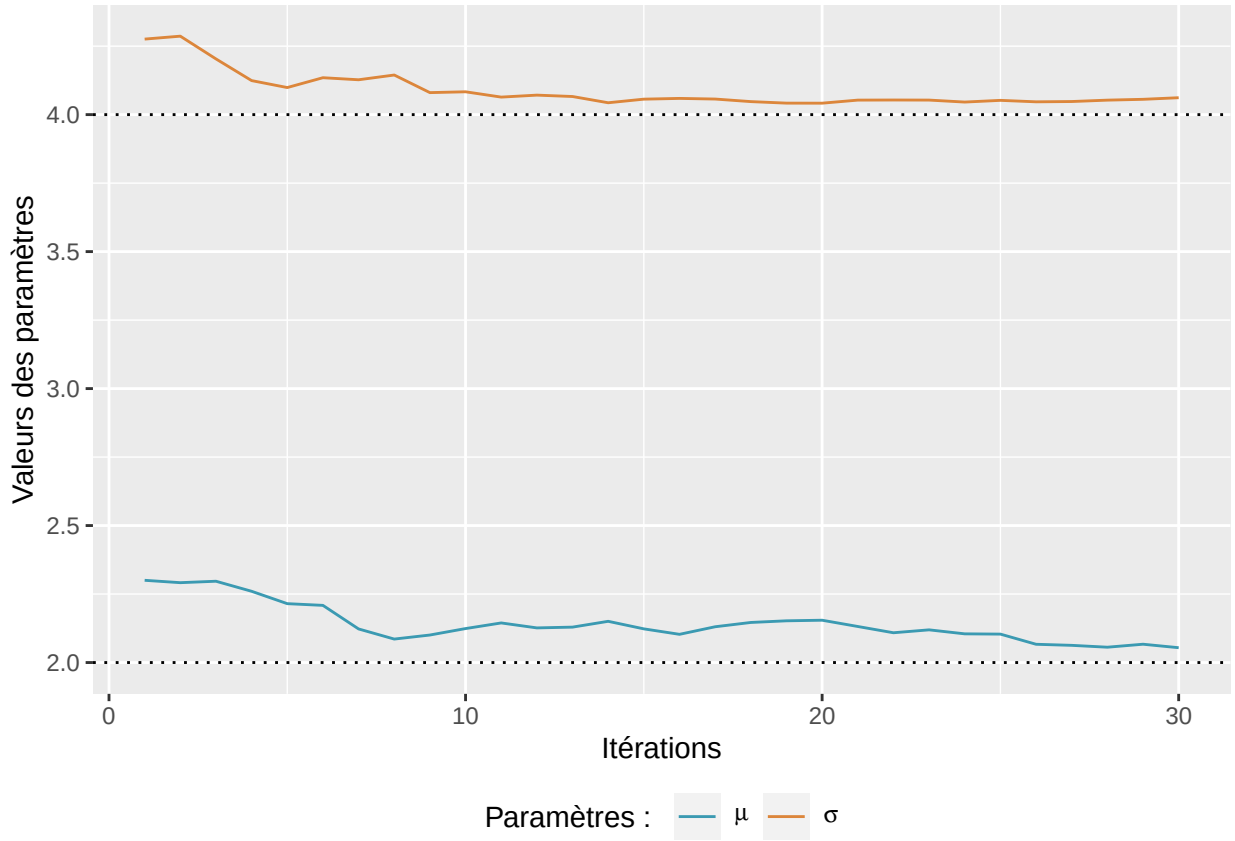
Nouvelle approche à venir.

```r
df_recurs <- read_csv("dataframes/df_recurs.csv")[,-1]
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   X1 = col_double(),
##   param_1 = col_double(),
##   param_2 = col_double(),
##   iteration = col_double()
## )
```

```r
colnames(df_recurs) = c("param_1", "param_2","iteration")

lab = list(bquote(mu), bquote(sigma))

#png('df_recurs.png', units="cm", width=15, height=11, res=300)

df_recurs_melted = melt(df_recurs, id.vars = "iteration")
ggplot(df_recurs_melted, aes(x = iteration, y = value)) + geom_line(aes(color = variable, group = variable
```



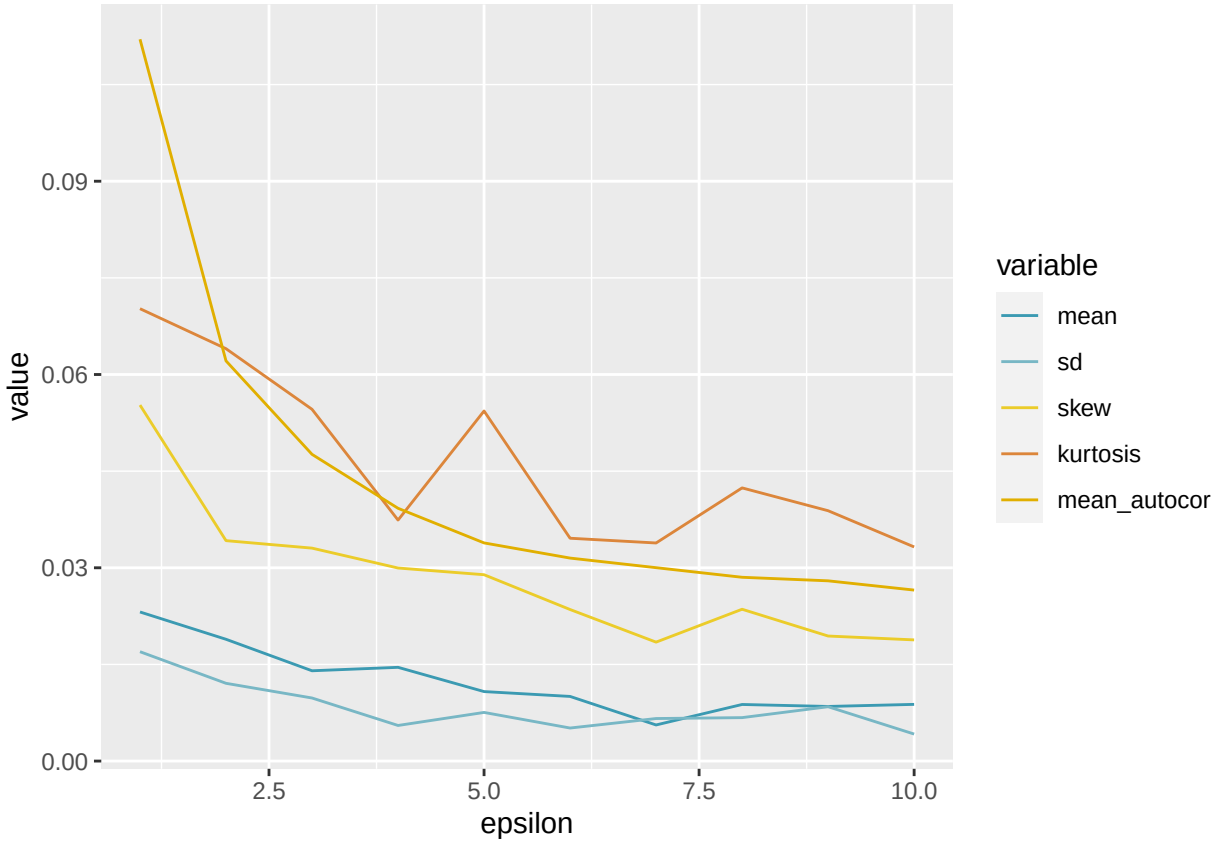

```r
#dev.off()
```

3.3 Hasting iid

Afin de pouvoir appliquer numériquement la méthode de Reverse logistic regression, on aurait besoin de savoir simuler un échantillon iid suivant une loi dont on ne connait pas la constante de normalisation. L'idée est d'utiliser l'algorithme d'Hasting et de ne conserver qu'un échantillon tous les ϵ pas. Le code est en haut de ce document.

Etude de l'impact du choix du pas.

```
ggplot(df_hasting_iid_melted, aes(x = epsilon, y = value)) + geom_line(aes(color = variable, group = va
```




```
#ggplot(df_hasting_iid, aes(x = epsilon, y = time)) + geom_line()
```

$\epsilon = 2$ semble être un bon choix au regard du gain en terme d'auto-corrélation et du temps de calcul.

3.4 Reverse logistic regression : deux lois de même famille

La maximisation de la fonction objectif

$$l_n(\eta) = \sum_{j=1}^{m} \sum_{i=1}^{n_j} log(p_j(X_{i,j}, \eta))$$

permet d'estimer les η (qui sont fonction des constantes de normalisation des h_j). On utilise les notations suivantes :

$$\eta_j = -log(Z_j) + log(\frac{n_j}{n}) \text{ avec } Z_j \text{ la constante de normalisation de } h_j$$

$$p_j(x) = \frac{h_j(x)e^{\eta_j}}{\sum_{k=1}^{m} h_k(x)e^{\eta_k}}$$

Exemple avec $m = 2$, $n = n_1 + n_2 = 1000 + 1000$, et pour coller avec les méthodes différentes on va prendre h_1 la densité non normalisée d'une $\mathcal{N}(\alpha)$ dont on a estimé α par MC MLE et h_2 la densité non normalisée d'une $\mathcal{N}(\psi)$ avec ψ qu'on choisit.

```
rev_log_reg = function(x, alpha, n, psi, h, eps){

  m = length(x)
  y = hasting_iid(x, n, psi, h, eps)

  # calcul des probabilités p_j
  denom = function(sample, eta) {
    return(pm_barre(sample, alpha)*exp(eta[1]) + pm_barre(sample,psi)*exp(eta[2]))}
  p_1 = function(sample, eta){
    return (pm_barre(sample, alpha)*exp(eta[1]) / denom(sample, eta))}
  p_2 = function(sample, eta){
    return (pm_barre(sample, psi)*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_1(x, eta))) + sum(log(p_2(y, eta))))}

  # initialisation descente de gradient
  eta1 = -log(sd(x)*sqrt(2*pi)) + log(m/(m+n))
  eta2 = -log(sd(y)*sqrt(2*pi)) + log(n/(m+n))

  # optimisation
  const = optim(
    par = c(eta1,eta2),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  a = exp(-const[1] + log(m/(m+n)))
  return(a)
}
```

```
pm_barre = function(u, theta){
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

m = 1000
n = 10000
x = rnorm(m,2,4)
psi = c(mean(x), sd(x))

alpha = mc_mle(x, n, psi, pm_barre)
print(alpha)
```

```
## [1] 1.490901 4.136059
```

```
print(rev_log_reg(x, alpha, n, c(8,8), pm_barre, 2))
```

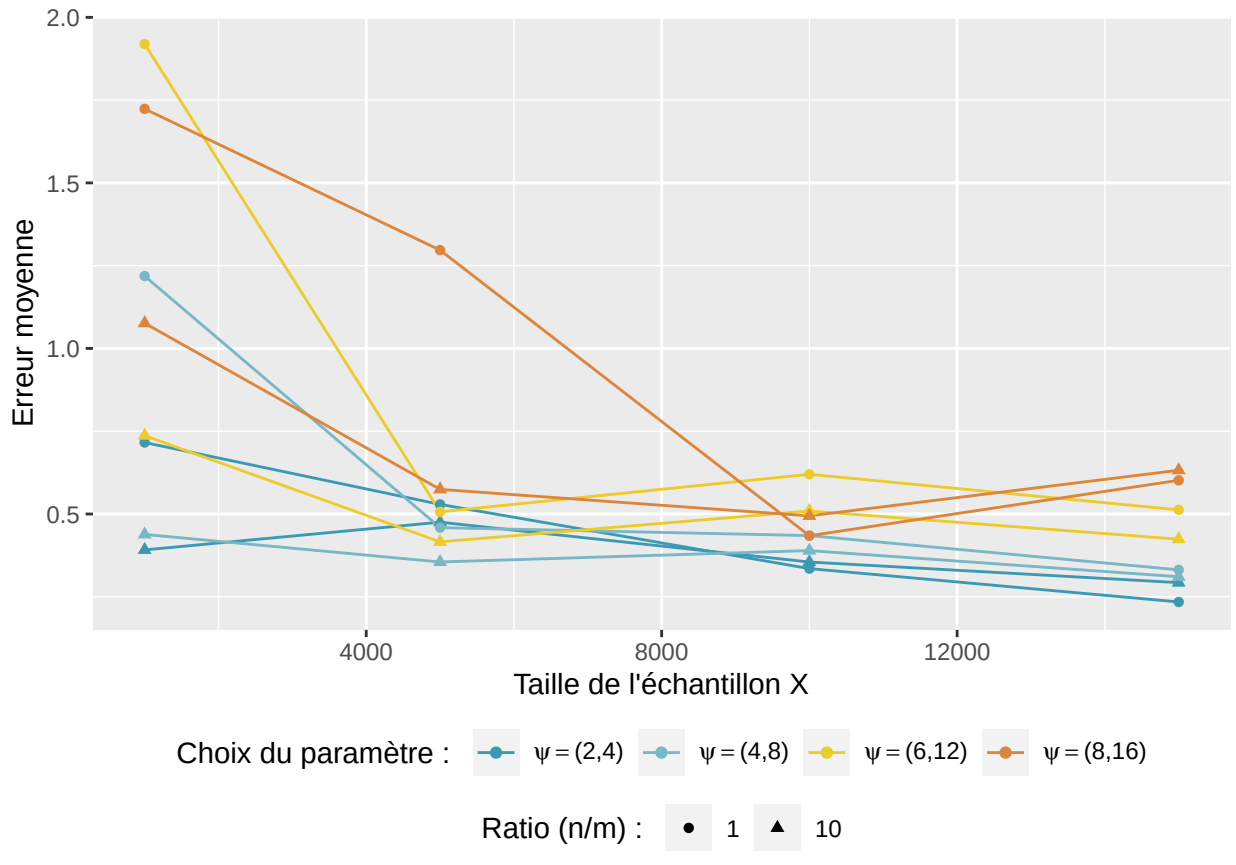
```
## [1] 10.01196
```

Etude de l'impact de la dimension, du ratio et des paramètres sur la convergence de la constante.

```
df_rev_log_reg <- read_csv("dataframes/df_rev_log_reg.csv")[,-1]
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
df_rev_log_reg_agg = aggregate(const ~ size_data + ratio_noise_data + ratio_alpha_psi,
                    data = df_rev_log_reg,
                    FUN = mean)
df_rev_log_reg_agg$ratio_alpha_psi = as.factor(df_rev_log_reg_agg$ratio_alpha_psi)
df_rev_log_reg_agg$ratio_noise_data = as.factor(df_rev_log_reg_agg$ratio_noise_data)
df_rev_log_reg_agg$size_data = as.numeric(df_rev_log_reg_agg$size_data)
df_rev_log_reg_agg$const_error = abs(df_rev_log_reg_agg$const - 4*sqrt(2*pi))

#png('df_rev_log_reg.png', units="cm", width=15, height=11, res=300)

ggplot(df_rev_log_reg_agg, aes(x = size_data, y = const_error, color = ratio_alpha_psi, shape = ratio_n
)
```




```
#dev.off()
```

Une tentative de la régression logistique inverse sur données gaussiennes, qui fonctionne bien, insensible aux conditions initiales.

```
rev_log_reg_gaussien = function(n1,n2,n3){

  pm1 = function(u){return(exp(-0.5 * ((u)**2)))} #connu
  pm2 = function(u,theta){return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))} #inconnu normalisation
  pm3 = function(u,theta){return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))} #inconnu normalisation
```

```
  x1 = rnorm(n1)
  x2 = rnorm(n2,5,3) # les paramètres sont (5,3) - psi
  x3 = rnorm(n3,-2,6) # chi
  n  = n1+n2+n3

  ## ATTENTION A BIEN MATCHER LES PARAMETRES PSI ET CHI AVEC LES TIRAGES X1,X2 ##
  psi = c(5,3)
  chi = c(-2,6)

  # calcul des probabilités p_j
  denom = function(sample, eta) {return(pm1(sample)*exp(eta[1]) + pm2(sample,psi)*exp(eta[2]) + pm3(sam

  p_1 = function(sample, eta){return (pm1(sample)*exp(eta[1]) / denom(sample, eta))}
  p_2 = function(sample, eta){return (pm2(sample, psi)*exp(eta[2]) / denom(sample, eta))}
  p_3 = function(sample, eta){return (pm3(sample, chi)*exp(eta[3]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta){return(sum(log(p_1(x1, eta))) + sum(log(p_2(x2, eta))) + sum(log(p_3(x3, eta))))}

  # initialisation descente de gradient
  eta1 =  -log(sd(x1)*sqrt(2*pi)) + log(n1/n) +100
  eta2 =  -log(sd(x2)*sqrt(2*pi)) + log(n2/n) +34
  eta3 =  -log(sd(x3)*sqrt(2*pi)) + log(n3/n) -45

  # optimisation
  eta = optim(par = c(eta1,eta2,eta3),gr = "CG",control = list(fnscale=-1),fn = L)$par

  constante_additive = eta[1] + log(sqrt(2*pi)) - log(n1/n)
  a = exp(-eta[1] + log(n1/n) + constante_additive) #valeur témoin / vrai valeur connue = 1
  b = exp(-eta[2] + log(n2/n) + constante_additive) # normlisation de linconnu
  c = exp(-eta[3] + log(n3/n) + constante_additive) # normlisation de linconnu

  x = c(a,b,c,constante_additive)

  return (x)
}


rev_log_reg_gaussien(10000,10000,10000)
```

```
## [1]  2.506628  7.527957 14.948548 15.306092
```

```
c(sqrt(2*pi*1),sqrt(2*pi*9),sqrt(2*pi*36)) # pour comparer a,b,c aux vraies constantes de normalisation
```

```
## [1]  2.506628  7.519885 15.039770
```

3.5 Reverse logistic regression : deux lois de familles différentes

On reprend les notations ci-dessus (notations du papier). Exemple avec $m = 2$, $n = n_1 + n_2$, et pour coller avec les méthodes différentes on va prendre h_1 la densité non normalisée d'une $\mathcal{N}(\alpha)$ dont on a estimé α par MC MLE et h_2 la densité d'une loi usuelle, on en connait donc la constante de normalisation. On note ψ le paramètre de cette loi usuelle.

```r
rev_log_reg_with_noise = function(x, law_noise, n, alpha, psi, h1, h2){

  y = do.call(law_noise, c(list(n),psi))
  m = length(x)

  # calcul des probabilités p_j
  denom = function(sample, eta) {
    return(h1(sample, alpha)*exp(eta[1]) + h2(sample,psi)*exp(eta[2]))}
  p_1 = function(sample, eta){
    return (h1(sample, alpha)*exp(eta[1]) / denom(sample, eta))}
  p_2 = function(sample, eta){
    return (h2(sample, psi)*exp(eta[2]) / denom(sample, eta))}

  # fonction objectif
  L = function(eta) {
    return(sum(log(p_1(x, eta))) + sum(log(p_2(y, eta))))}

  # initialisation descente de gradient
  eta1 = -log(sd(x)*sqrt(2*pi)) + log(m/(m+n))
  eta2 = -log(sd(y)*sqrt(2*pi)) + log(n/(m+n))

  # optimisation
  const = optim(
    par = c(eta1,eta2),
    gr = "CG",
    control = list(fnscale=-1),
    fn = L
  )$par

  constante_additive = const[1] + log(512) - log(1/2)

  b = exp(-const[2] + log(n/(m+n)) + constante_additive)
  a = exp(-const[1] + log(m/(m+n)) + constante_additive)

  return(c(a,b,const[1],const[2], log(0.5), log(512) - log(0.5), constante_additive))
}
```

```r
pm_barre = function(u, theta){
  return(exp(-0.5 * ((u - theta[1]) / theta[2]) ** 2))
}

pn_cauchy = function(u, psi){
  return(dcauchy(u, psi[1], psi[2]))
}

pn_norm = function(u, psi){
  return(dnorm(u, psi[1], psi[2]))
}

pn_unif = function(u, psi){
  return(dunif(u, psi[1], psi[2]))
}

m = 1000
```

```
n = 10000
x = rnorm(m,2,4)
psi = c(mean(x), sd(x))

#alpha = mc_mle(x, n, psi, pm_barre)
```

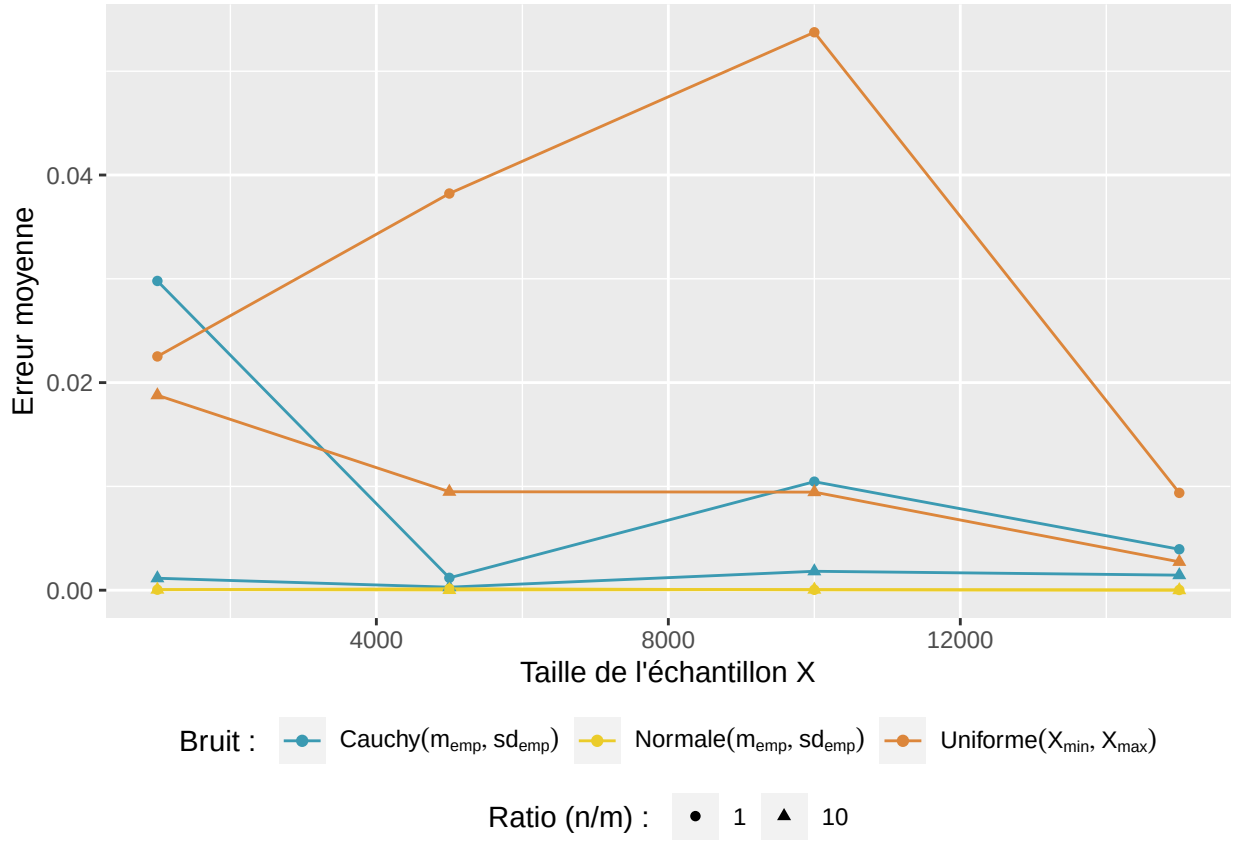
```
rev_log_reg_with_noise(x, rcauchy, n, c(2,4), c(mean(x),sd(x)), pm_barre, pn_cauchy)
```

```
## [1] 93.0909091  9.2749187 -8.0945173 -3.4856695 -0.6931472  6.9314718 -1.1630455
```

```
df_rev_log_reg_noise <- read_csv("dataframes/df_rev_log_reg_noise.csv")[,-1]
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
df_rev_log_reg_agg = aggregate(const ~ size_data + ratio_noise_data + law_noise,
                    data = df_rev_log_reg_noise,
                    FUN = mean)
df_rev_log_reg_agg$ratio_noise_data = as.factor(df_rev_log_reg_agg$ratio_noise_data)
df_rev_log_reg_agg$size_data = as.numeric(df_rev_log_reg_agg$size_data)
df_rev_log_reg_agg$const_error = abs(df_rev_log_reg_agg$const - 4*sqrt(2*pi))

#png('df_rev_log_reg_noise.png', units="cm", width=15, height=11, res=300)

ggplot(df_rev_log_reg_agg, aes(x = size_data, y = const_error, color = law_noise, shape = ratio_noise_da
```




```
#dev.off()
```

4 Application : Modèle d'Ising

4.1 Simulation en 2D

Algorithme de Gibbs : on part d'une configuration aléatoire. A chaque itération, on sélectionne un site σ_j au hasard. Ce site prend la valeur 1 avec probabilité $p := \mathbb{P}(\sigma_j = 1|\sigma_{-j})$ et -1 avec probabilité $1 - p$. σ_{-j} est une notation qui désigne les voisins de σ_j.

On a $p = \frac{1}{1+exp(\beta \Delta H)}$ avec $\Delta H = 2\sigma_j \sum_j \sigma_{-j}$.

```
# Simuler une matrice uniforme
sim_unif_2D = function(height, width){
  return(matrix(sample(c(-1,1), height * width, replace=T), ncol = width))
}

# Empiler verticalement plusieurs matrices uniformes
samples_unif_2D = function(n, height, width){
  return(sim_unif_2D(n * height, width))
}

# Sélection des voisins, dans l'ordre haut - droite - bas - gauche

get_neighbors = function(sample, row_site, col_site){
  height = dim(sample)[1]
  width = dim(sample)[2]

  if (height == 1){
    up = 0
    down = 0
  }
  else if (row_site == 1){
    up = 0
    down = sample[row_site + 1,col_site]
  }
  else if (row_site == height) {
    up = sample[row_site - 1,col_site]
    down = 0
  }
  else {
    up = sample[row_site - 1,col_site]
    down = sample[row_site + 1,col_site]
  }
  if (width == 1){
    left = 0
    right = 0
  }
  else if (col_site == 1){
    left = 0
    right = sample[row_site, col_site + 1]
  }
  else if (col_site == width) {
    left = sample[row_site, col_site - 1]
    right = 0
  }
  else {
```

```
    left = sample[row_site, col_site - 1]
    right = sample[row_site, col_site + 1]
  }

  return(c(up, right, down, left))
}

# Calcul de l'énergie

compute_energy = function(sample, height, width){
  energy = 0
  for (i in 1:height){
    for (j in 1:width){
      energy = energy - sample[i,j]*sum(get_neighbors(sample, i, j))
    }
  }
  return(energy/2)
}

compute_delta_energie = function(sample, row_site, col_site){
  spin = sample[row_site, col_site]
  return(2 * spin * sum(get_neighbors(sample, row_site, col_site)))
}

# Simulation

sim_ising_2D = function(beta, height, width, iter, init) {
  config = init
  for (i in 1:iter) {
    row = sample(1:height, 1)
    col = sample(1:width, 1)
    p = 1/(1 + exp(beta * compute_delta_energie(config, row, col)))
    config[row, col] = sample(c(1,-1), 1, prob = c(p, 1-p))
  }
  return(config)
}

# Graphique

graph_config = function(config){
  height = dim(config)[1]
  width = dim(config)[2]
  colnames(config) <- paste("Col", 1:width)
  rownames(config) <- paste("Row", 1:height)
  df <- melt(t(config))
  colnames(df) <- c("x", "y", "value")

  return(ggplot(df, aes(x = x, y = y, fill = factor(value)))
         + geom_tile()
         + coord_fixed()
         + scale_fill_manual(values=pal2_bis)
         + labs(col="Spins : ")
         + theme(axis.ticks.x = element_blank(), axis.text.x = element_blank(), axis.ticks.y = element_
```

```
        + theme(legend.position = "none")
        )
}
```

On va générer un échantillon de m configurations distribuées selon un modèle d'Ising en 2D. Le résultat sera présenté sous la forme d'une matrice verticale où seront empilées les configurations.

```
samples_ising_2D = function(beta, height, width, burn_in, epsilon, m){
  # beta : paramètre du modèle
  # height, width : taille d'une configuration
  # burn_in : nombre d'itérations pour atteindre la stationnarité
  # epsilon : pas de décorrélation pour sélectionner des configs iid
  # m : nombre de configurations dans notre échantillon

  # initialisation
  current_config = sim_unif_2D(height, width)

  # On effectue des itérations jusqu'à atteindre supposément l'état stationnaire
  current_config = sim_ising_2D(beta, height, width, burn_in, current_config)
  samples = current_config

  # Une fois la stationnarité atteinte, on prend un échantillon tous les epsilon pas afin que nos échan
  for (i in 2:m){
    current_config = sim_ising_2D(beta, height, width, epsilon, current_config)
    samples = rbind(samples, current_config)
  }

  return(samples)
}
```

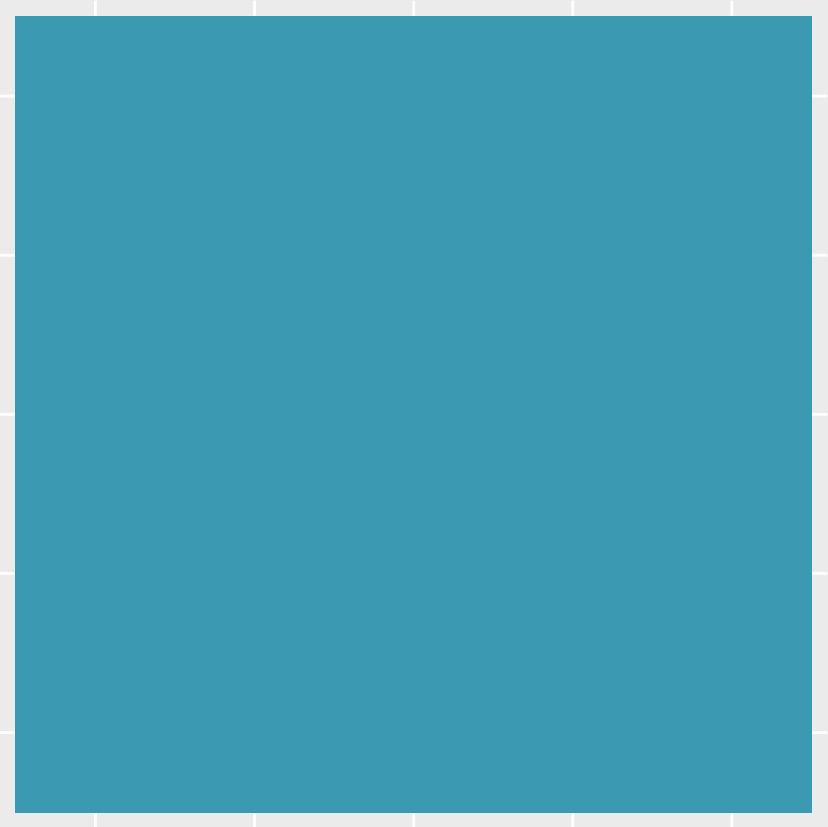
On définit les fonctions p_n, p_m, etc, pour l'application des méthodes MCMLE, Régression logistique inverse, NCE.

```
pm_barre_ising = function(samples, alpha){
  m = dim(samples)[1] / height
  energie_for_each_sample = c()
  for (i in 0:(m-1)){
    id_beginning_sample = 1+i*height
    id_end_sample = id_beginning_sample + height - 1
    energie_for_each_sample = append(energie_for_each_sample, compute_energy(samples[id_beginning_sampl
  }
  return(exp(-alpha * energie_for_each_sample))
}

log_pm_ising = function(samples, theta){
  # theta[1] = beta
  # theta[2] = -log(Z)
  m = dim(samples)[1] / height
  energie_for_each_sample = c()
  for (i in 0:(m-1)){
    id_beginning_sample = 1+i*height
    id_end_sample = id_beginning_sample + height - 1
    energie_for_each_sample = append(energie_for_each_sample, compute_energy(samples[id_beginning_sampl
  }
  return(-theta[1] * energie_for_each_sample + theta[2])
```

```
}

pn_discr_unif_2D = function(samples, param){
  # param[1] = height
  # param[2] = width
  samples = matrix(t(samples), ncol = param[1] * param[2], byrow = TRUE)
  return(rowProds(ifelse(samples == -1, 1/2, ifelse(samples == 1, 1/2, 0))))
  #return(1/(2)^(param[1]*param[2]))
}

log_pn_discr_unif_2D = function(samples){
  return(log(pn_discr_unif_2D(samples, c(height, width))))
}
```

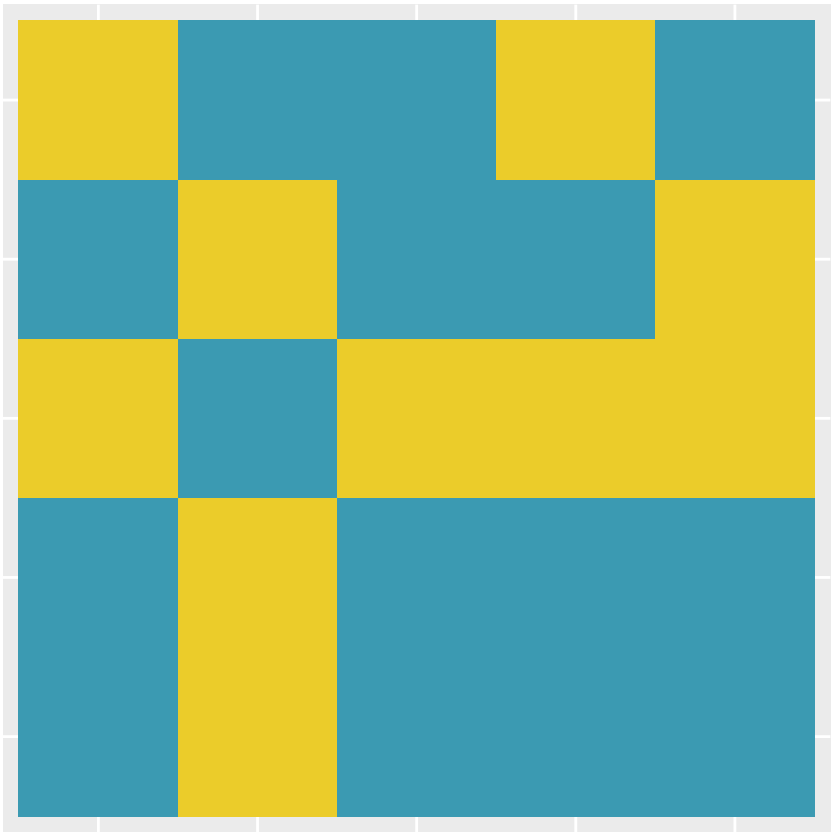
```
#---- Constantes pour l'application numérique ----#
beta = 0.5
psi = 0.5
width = 5
height = 5
burn_in = 100000
epsilon = 100
m = 1000
n = 1000
x = samples_ising_2D(beta, height, width, burn_in, epsilon, m)
y = samples_ising_2D(psi, height, width, burn_in, epsilon, n)
```

```
# Graphiques pour visualiser l'évolution de la chaine de markov durant la période de chauffe


#png('ising_init.png', units="cm", width=7, height=4, res=300)
test1 = matrix(rep(-1, height*width), ncol = width)
graph_config(test1)
```

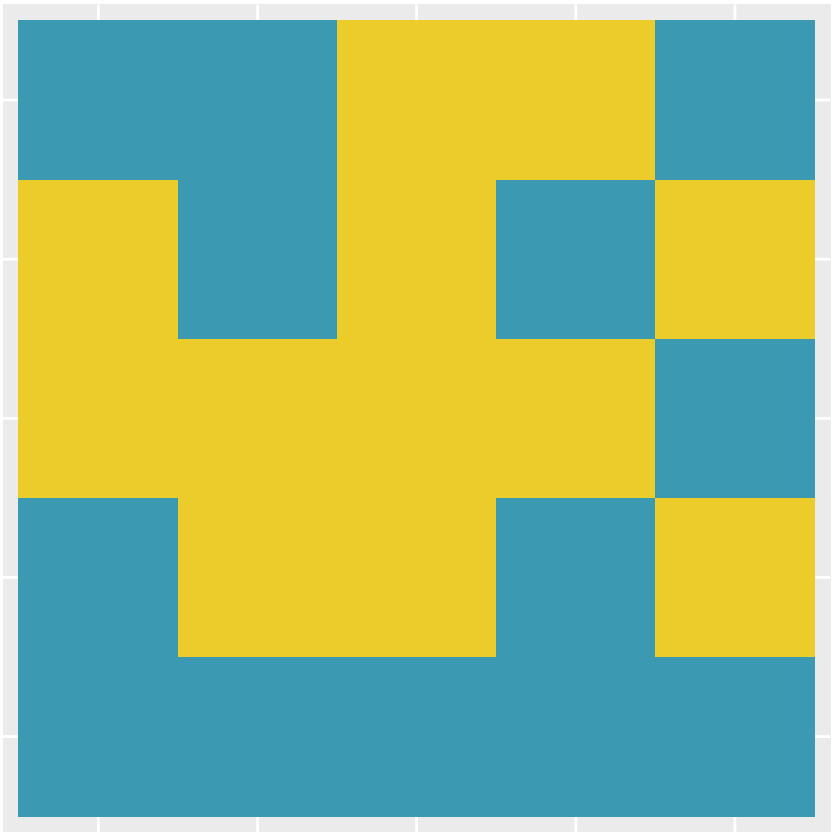


```
#dev.off()

#png('ising_iter100.png', units="cm", width=7, height=4, res=300)
test2 = sim_ising_2D(beta, height, width, 100, test1)
graph_config(test2)
```

```
#dev.off()

#png('ising_iter1000.png', units="cm", width=7, height=4, res=300)
test3 = sim_ising_2D(beta, height, width, 900, test2)
graph_config(test3)
```



```
#dev.off()

#png('ising_iter10000.png', units="cm", width=7, height=4, res=300)
test4 = sim_ising_2D(beta, height, width, 9000, test3)
graph_config(test4)
```



```
#dev.off()

#png('ising_iter100000.png', units="cm", width=7, height=4, res=300)
test5 = sim_ising_2D(beta, height, width, 100000, test4)
graph_config(test5)
```

```
#dev.off()
```

```
#------- MC MLE ---------------------------------------------------------------------#

# Fonction objectif
L = function(alpha){ return(sum(log(pm_barre_ising(x,alpha)/pm_barre_ising(x,psi))) - m*log(mean(pm_bar

optimize(L, c(0,1), maximum = TRUE)$maximum
```

```
#------ Régression logistique inverse ------------------------------------------------#

# calcul des probabilités p_j
denom = function(sample, eta) {
  return(pm_barre_ising(sample, beta)*exp(eta[1]) + pn_discr_unif_2D(sample,c(height, width))*exp(eta[2
p_1 = function(sample, eta){
  return (pm_barre_ising(sample, beta)*exp(eta[1]) / denom(sample, eta))}
p_2 = function(sample, eta){
  return (pn_discr_unif_2D(sample, c(height, width))*exp(eta[2]) / denom(sample, eta))}

# fonction objectif
L = function(eta) {
  return(sum(log(p_1(x, eta))) + sum(log(p_2(y, eta))))}

# initialisation descente de gradient
eta1 = -log(sd(x)*sqrt(2*pi)) + log(m/(m+n))
eta2 = -log(sd(y)*sqrt(2*pi)) + log(n/(m+n))
```

```r
# optimisation
const = optim(
  par = c(eta1,eta2),
  gr = "CG",
  control = list(fnscale=-1),
  fn = L
)$par

# calcul de la constante
constante_additive = const[2] + log(1) - log(n/(m+n))
exp(-const[1] + log(m/(m+n)) + constante_additive)
```

```r
# ------- NCE ---------------------------------------------------------------------------#

h = function(u, theta){
    return( 1 / (1 + n/m * exp(log_pn_discr_unif_2D(u) - log_pm_ising(u, theta))))
  }

  J = function(theta){
    return( sum(log(h(x, theta))) + sum(log(1 - h(y, theta))) )
  }

  theta = optim(
    par = c(0,-138),
    gr = "CG",
    control = list(fnscale=-1),
    fn = J
  )$par

  print(c(theta[1], exp(-theta[2])))
```

```r
# Constante calculée par le package GiRaf
NC.mrf(height, width, beta)
```